Access Group Design

Goal

Allow providers and system administrators to create groups of CMR users so that they can collectively be assigned permission to various actions and entities in the CMR.

Traceability

CMR-2139 - JIRA project doesn't exist or you don't have permission to view it.

Access Control Group Epic

Group Representation

Groups contain the following fields

- name
  - Contains the name of the group. Name is unique in combination with provider id.
  - String of max length 100
  - May not contain ASCII group separate character
- provider-id
  - Identifies which provider owns the group or if nil it is a system owned group
- description
  - String of max length 255 describing the group.
- members
  - A set of URS user names
- legacy-guid
  - Optional used only for legacy integration with the kernel.

Groups will be stored in Metadata DB as a new concept type.

- Native id is group name.
  - Groups are uniquely identified by group name and provider id. Provider id will be used in the metadata db concept like collections and granules for uniqueness.
- Concept id letter is AG which stands for access control group.
- concept type is :access-group
- Metadata is stored as EDN

Example Metadata DB concept for a provider level group

```
{:concept-type :access-group
:native-id "Administrators"
:concept-id "AG1-LPDAAC_ECS"
:provider-id "LPDAAC_ECS"
:user-id "user101"
:metadata "..." ;; edn as described below
:format "application/edn"
:revision-id" 1
:revision-date" "2012-01-01T00:00:00"}
```

Example Metadata DB concept for a system level group

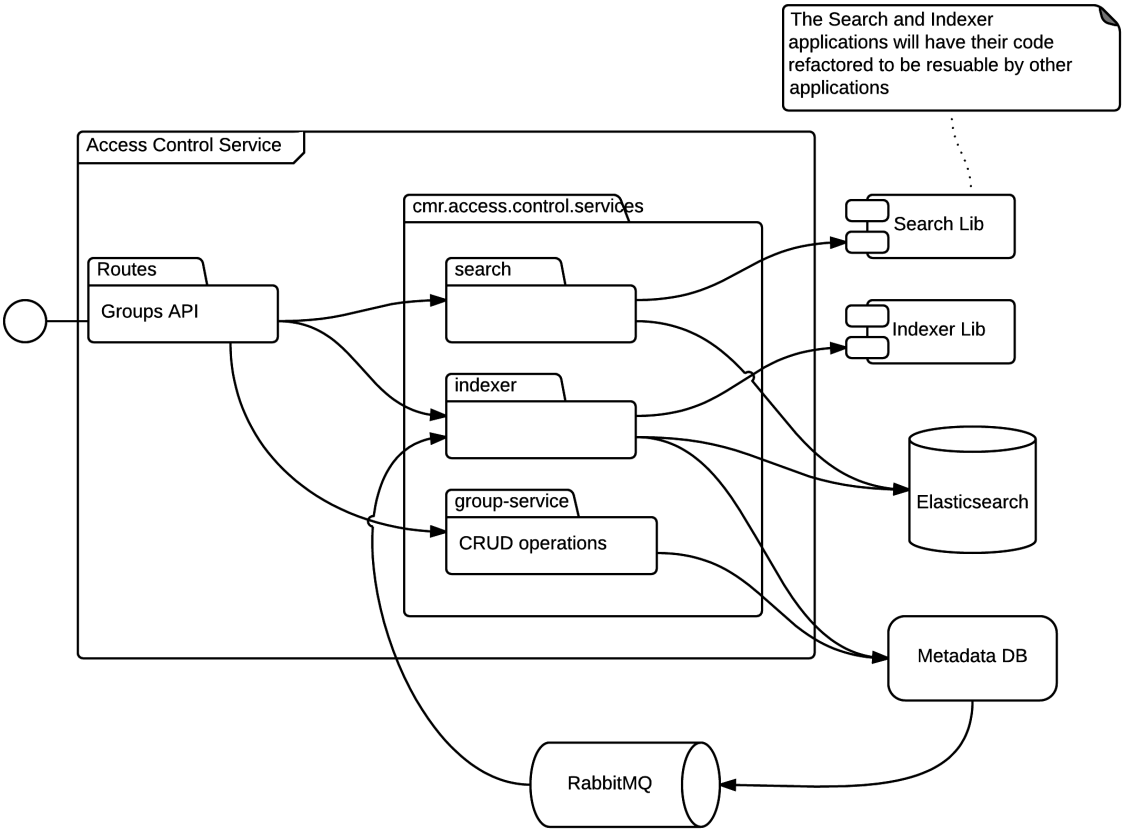
```
{:concept-type :access-group
:native-id "Administrators"
:concept-id "AG1-CMR"
:provider-id :cmr
:user-id "user101"
:metadata "..." ;; edn as described below
:format "application/edn"
:revision-id" 1
:revision-date" "2012-01-01T00:00:00"}
```

Example metadata in concept map

```
{:name "Administrators"
:provider-id "LPDAAC_ECS"
:description "Contains users with permission to manage LPDAAC_ECS holdings."
:members ["jsmith" "prevere" "ndrew"]}
```

Access Control Service Design

Groups, tokens, and access control lists will all be managed on a new CMR service called Access Control.



Designed to be cohesive and have low coupling with the rest of the CMR

The CMR is currently divided into functional components like Indexer, Search, Ingest, and Metadata DB. Current design downsides:

- This design results in more coupling and less cohesion.
- Developers run the entire CMR. Takes a long time for REPL to start.
- Changes often impact multiple services.
- All tests in CMR are run even though a developer may be modifying a small portion of the CMR.

The Access Control Service starts a new design trend in the CMR where microservices are split by concept type. Splitting by concept type has several ramifications.

- A single application is responsible for the entire lifecycle of a set of concept types
- Common capability code (ingest, search, indexing) is shared through libraries. Code from the existing Search and Indexer applications will be refactored so that the common parts will be in a library.

The benefits of this approach are:

- It allows a developer to focus on a given concept type without running the entire CMR.
- All the code for a concept type will be located in a small area. It should be easy to see in a couple of namespaces how we validate groups on ingest, how we index them, and how we search them.
- The common search and indexing code is factored out meaning there's less generic stuff along with the concept type code.
- Less coupling between CMR services
    - The access control service will only be accessed from other components
- Faster REPL startup time
- Smaller set of tests to run.

It doesn't preclude running all of the CMR as a single application like it is now. Developers can do this if they're changing some of the core

common search and indexing code or making another kind of overall change. Developers don't have to run all tests in the entire CMR. CI will run all the tests in the CMR for their branch before it is merged.

Comments on Access Control Service Design

These are some notes on issues identified by developers during a design review of the access control service above. The comments were captured after the design review by me (Jason) from memory so I may not have capture the entire points.

There's a loop in the information flow above

Changes from groups are saved to metadata db. Metadata db generates messages which are then asynchronously handled by group indexing. We could almost drop that and just do the indexing once the data has been saved.

The Rabbit MQ configuration will grow in complexity.

We'll probably have message queues per concept type.

Microservices add overhead to development and make things more complicated.

The reviewer indicated that you have to consider all of the different services during development. When you want to make request over HTTP that's harder to do. It's not as simple as a regular function call. You have to jump from service to service to understand how the different pieces connect together. This increase the net complexity and increases the amount you have to think about during development of the CMR.

This is a great criticism of the current state of microservices as used in the CMR. The goal of microservices is to *decrease complexity* at the individual service level and make it easier to develop. The current microservice structure fails to achieve that for the most part. I noted above that most developers have to run the entire CMR. When microservices are designed correctly they are:

- cohesive - They gather together all the things that should go together and exclude everything else.
- low coupling - They can change independently. They aren't chatty.

If we had achieved that a CMR developer would be able to run a single service and mostly change code in just that service for a single issue. The fact that you mostly don't do that now is a sign that the CMR microservices aren't cohesive and low coupled. The code for handling collections is spread out among ingest, umm, search, and indexer. There are two services and some other places where we do achieve this which are metadata db and cubby. Both of those services can be developed by themselves. They're mostly fairly simple and have a simple api. It's easy to develop them.

The goal of the new access control service is to fix those problems and achieve the same kind of easy development that metadata db and cubby have. I think we can show that it will by answering a couple of questions.

Question: How many services will have to deal with groups, acls, or tokens? How often will access control service be invoked from other parts of the CMR?

Answer: The access control service should be the only service that cares about groups and acls. Other services will receive a token as part of a request. They only care about knowing two things: 1. Is this a valid token? and 2. Does the user identified with this token have permission to do something? The other services should be able to invoke one api function on access control taking a token and information identifying the action and object they want to modify. The access control service will take care of the rest including what groups is this user apart of? What ACLs are related to those groups? Do any of the ACLs grant permission to the specified action?

An exception to that is in the search app for how it applies ACLs. It has to translate them into query conditions in some cases so it will need to retrieve ACLs related to the current user. It should be able to do this by requesting ACLs for catalog items applicable to the current user.

Does the CMR require microservices? Could we go to a more monolithic architecture?

The CMR could work without microservices right now. It could be a single monolithic application and would technically work. We'd lose some abilities if we do this. Some of the capabilities we'd lose are more forward looking things. ECHO used a monolithic architecture and eventually became a quagmire unable to evolve technologically. These are a few of the capabilities that would be more difficult as a monolith.

- Individual scaling of individual parts. We don't current take advantage of this capability now but this will become more relevant as the CMR increases in scale. We may start getting a lot more search requests as users increase or clients get more complex. New data like Sentinel is supposed to bring a billion new granules.
- Ability to focus on individual components during development.
- Add new capabilities that aren't coupled with current ones. Virtual products is a good example of a low coupled capability that was added after the fact. We created a new microservice with minimal changes to ingest and search.
- Change technologies over time.

ElasticSearch Index

The ElasticSearch index settings and mappings will be defined in the access control service not the index set application. The index set application hasn't turned out to be useful and will probably be removed eventually. Updating the index to the latest mappings will also be done in the access control group. This will be done in a very similar way to the Cubby application. See how Cubby defines the mappings and database migrations.

Group API Design

- /groups
    - POST - create create
    - GET - search for groups
        - Search would be implemented via elasticsearch
        - Fields - provider id, member, legacy guid, concept id
        - Returns full group representation with list of members.
    - /:concept-id
        - PUT - update the group
        - DELETE - delete the group
        - GET - get the group
        - /members
            - POST - Add new members. Takes a json array of members to add.
            - DELETE - Removes members. Takes a json array of members to add.
    - /translate - POST
        - Access a list of group guid and returns a list of group concept ids.
        - Uses elasticsearch for performance
        - Must return nils of group guid doesn't exist
        - Concept ids can be passed in as a guid. (allows us to switch to concept ids at some point and return as guids)
    - /reindex - POST
        - Reindexes all of the groups asynchronously
- Other standard routes like /health and /reset

Questions

- I'm most unsure about the members in groups. The kernel returns all of them with the group. Should we do that in the CMR? I think normal use cases for groups don't have that many users.

Kernel Implementation

Group2ManagementService

There will be two implementations of this interface. One will be a switcher that changes based on a kernel property whether to make make the kernel primary or the CMR primary. The switcher will use parallel fanout except where noted. The other implementation will invoke the CMR via REST. The switcher will delegate to the original kernel implementation and the CMR implementation

String createGroup(Group2BO group, String managingGroupGuid, ServiceContext context)

Switcher Implementation

The creation process does not use double parallel fanout. It always invokes the kernel first so that the GUID is captured and then it is sent to the CMR. If the CMR is primary the response is used. If it is not the primary it is invoked in a separate thread after kernel create and the response is thrown away.
managingGroupGuid will not be sent to the CMR. The CMR will not create the ACL necessary. We'll rely on the kernel doing that for now. Before we can turn off the kernel first part of creating a group we'll need to have migrated ACLs to the CMR and added this capability to the group service

CMR Implementation

1. Send the request to the Kernel and capture guid in created group (if successful)
2. CMR Creation
    a. Get token from the context
    b. Translate provider guid to provider id
    c. Translate member guids to URS usernames

    d. Send request to the CMR to create the group
3. Return the guid captured from the kernel

void removeGroup(String groupGuid, ServiceContext context)

CMR Implementation

1. Translate group guid to group concept id.
2. Get token from context
3. send DELETE to /groups/:concept-id

void updateGroup(Group2BO group, ServiceContext context)

CMR Implementation

1. Translate group guid to group concept id.
2. Get token from context
3. Translate group to CMR representation
4. PUT group to /users/:concept-id

void addUserToGroup(Group2BO group, String userGuid, ServiceContext context)

CMR Implementation

1. Translate group guid to group concept id.
2. Get token from context
3. POST user id to /groups/:concept-id/members

NameGuidBO[] getGroupNamesByOwner(String providerGuid, ServiceContext theContext)

CMR Implementation

1. Translate provider guid to provider id
2. GET /groups?provider-id=<provider-id>
3. Return legacy guid and group name

NameGuidBO[] getGroupNamesByMember(String memberGuid, ServiceContext theContext)

CMR Implementation

1. Translate user guid to urs name
2. GET /groups?member=<urs user>
3. Return legacy guid and group name

Group2BO[] getGroups(List<String> groupGuids, ServiceContext theContext)

CMR Implementation

1. GET /groups?legacy-guid[]=<legacy guids>
2. Return groups translated from CMR

void notifyGroup(String groupGuid, String message, ServiceContext theContext)

Throws unsupported operation exception

void removeReferences(UserBO user)

Does nothing. Ignores calls

Group2MemberDao

The AuthenticationService calls getGroupsSidsByUserGuid so we must implement a switcher and implementation for this as well. We can piggyback this single method on the other implementations.

CMR Implementation

1. Translate user guid to URS user name
2. Use the system read token as the token since there is no context.
3. GET /groups?member=<urs user>
4. Return a list of the legacy group guids.

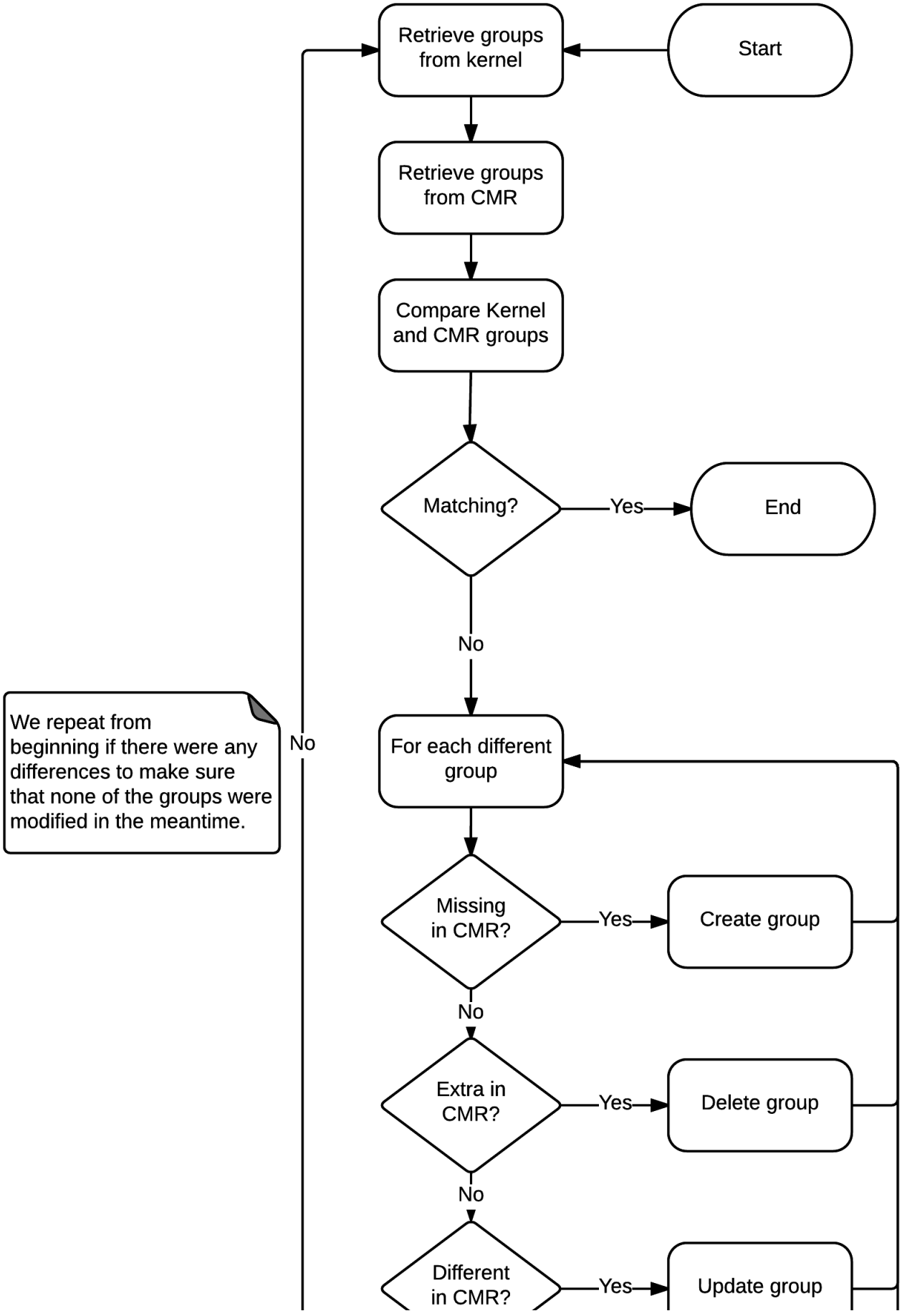
Kernel Testing

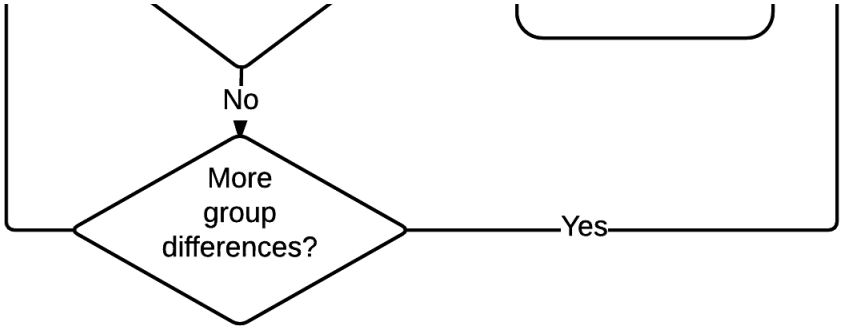
We need to be able to integration test our CMR integration code with the kernel. We will run the Kernel in the CMR CI by building and running the CMR as a simple Java application with Jetty. We'll need to define a project containing kernel integration tests. We should be able to run tests with the switcher in different configurations to see if it continues to work. The Kernel uses Oracle which is supported in CI now in a docker container. We'll need to create the tables and bootstrap necessary data to get this working.

Bootstrap

We typically bootstrap things in the Bootstrap application. We do that because synchronizing data is tricky and you want it to be performance efficient. Performance is less of a concern with groups because it contains less data. We'll write bootstrap in the Access Control service itself.

We'll pull data from the kernel using the SOAP API. We'll retrieve all of the groups from the Kernel and all of the latest groups in the access control service and compare. The following flow chart shows how we'll do this:


Start
Retrieve groups from kernel
Retrieve groups from CMR
Compare Kernel and CMR groups
Matching?
Yes
End
No
For each different group
Missing in CMR?
Yes
Create group
No
Extra in CMR?
Yes
Delete group
No
Different in CMR?
Yes
Update group
No
We repeat from beginning if there were any differences to make sure that none of the groups were modified in the meantime.



Error rendering macro 'pageapproval' : null